AN INTERACTIVE PROGRAM MUTATION SYSTEM

A THESIS
Presented to
The Faculty of the Division of Graduate Studies

By
Daniel M. St.Andre'

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in
Information and Computer Science

Georgia Institute of Technology
December, 1978

AN INTERACTIVE

PROGRAM MUTATION SYSTEM

Approved:

Richard A. DeMillo, Chairman

Philip H. Enslow

Alton P. Jensen

ABSTRACT

An overview of the theory of program mutation is presented and the operation of a pilot system for using program mutation on FORTRAN subroutines is described. This pilot system is known as PIMS. The subroutines accepted by PIMS are limited to a subset of the FORTRAN programming language. The internal operation of PIMS is discussed in general terms and the associated user interactions are also described. Finally, some observations on the utilization of PIMS are reported.

ACKNOWLEDGEMENTS

I would like to thank the members of our reading committee, R.A. DeMillo, chairman, P.H. Enslow, Jr., and A.P. Jensen for their guidance and patience; F. Sayward, Yale University, for his contributions to my understanding of the goals being sought; and T. Budd, University of California at Berkley, for his illumination of the road ahead.

My special appreciation goes to A. Acree and J. Burns, Georgia Tech, for their help, criticism and dedication. Without their effort my work would not have been as successful.

This project was funded in part by Army Research Office Grant DAAG29-76-6-0338(G36-617).

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

This paper is a discussion of the operation and implementation of a pilot system for using program mutation on FORTRAN subroutines. Since this is an operational discussion, I will not attempt a detailed theoretical study of mutation analysis. Such studies are available in [14,15]. I will discuss the use of program mutation as a means for testing program correctness. The pilot system will accept an input file which is assumed to contain a FORTRAN subroutine valid in the language subset (see Appendix B). Mutations are generated according to operator commands and each mutant is checked for correctness.

A familiarity with the PRIMOS operating system and file management system, as applied to the PRIME-400 computer, and a familiarity with the Software Tools Subsystem [17] is assumed. Detailed discussion of the respective command syntax will be avoided except where required for clarity or completeness.

The pilot system, PIMS, is divided into three operational phases: a pre-run phase, a mutation phase, and a post-run phase. In the pre-run phase files are opened or created and instructions are accepted for processing in other phases. If an "internal-form" file does not exist when the pre-run phase begins, a run is called an initial

run. Throughout all phases of system operation, activities are different for initial and subsequent runs.

In the mutation phase, mutants are created and tested. It is the mutation phase that is the central part of the system. The subroutine that was submitted and parsed during the pre-run phase, provides an Internal Form File for use in the creation of mutants. It is in the mutation phase that the tuples of the internal form are modified and placed in the Mutant Information File for use during mutant execution. After all desired mutants have been generated, those specified are then retrieved and executed by the PIMS interpreter.

During the post-run phase, statistics are displayed about the mutations tested thus far in the processing. In addition, files are closed for use in subsequent runs. These subsequent runs merely repeat the same three phases already used until the user is satisfied with the results obtained. In short, the user will repeat the three phases until all "live" mutants are "killed" either through the development of a suitable test case or the determination by the user that one or more live mutants are equivalent to mutants already killed.

One feature of the system's design is the ability to execute the three phases in sequence and then loop back to the pre-run phase for additional processing. In this manner, a user can operate the system and gain insight which

is then used in tailoring the responses in the next pass. This repetitive refinement of the test data contributes greatly to the rapid convergence on a set of acceptable test data[14].

In the present implementation, the entire system is resident in approximately 67K words of virtual memory. The phases are independent enough procedurally that they may be overlayed for use on smaller memory configurations.

CHAPTER II

THEORETICAL OVERVIEW

An increasing productivity burden on software developers has contributed to the increased use of automated aids for the design, implementation and debugging of large-scale software products. However, these aids are intended for the actual programmers and first-level management.[16] They provide qualitative descriptions rather than quantitative information that may be used throughout a management hierarchy. The typical manager will ask questions like, "How close is the project to something that the users will find acceptable as a first release?" and, "How well has the program been tested?" The techniques of modularization[14], structured programming[14], and program verification[9,10] do not seem to answer these and similar questions. This lack of answers appears understandable because management should not be expected to understand programming languages and/or sophisticated mathematics. In this chapter, we will explain how a new testing approach known as program mutation[15] can be used to manage software effectively.

A statement of the program testing problem, as seen by management, might be: Given a program module and its associated test data, how well does the data test the module--in quantitative terms? To solve this problem,

program mutation provides a quantitative measure of the "goodness" of the test data. We make the assumption that the better the test data (i.e., the more complete it is), the more thorough a program has been tested. And in a different fashion, the more thorough the testing, the more confidence can be placed in a program's correctness.

A pilot system which performs program mutation[15] produces a "score" which indicates the adequacy of the test data. The users attempt to improve upon this "score" by either augmenting the test data or by answering "questions" about the program being tested. These questions address the essence of the program by forcing the user to compare alternate forms of a given statement and to make a decision whether the many forms are equivalent. This process of supplying test data, answering questions and interpreting results continues interactively until the user is satisfied with the quality of the test data. Meanwhile, all of the data remains available at each iteration for management review so that a quantitative answer to "how well a program has been tested" can be obtained.

Mutation Methodology

Program testing cannot be deductive. We know this since program testing attempts to derive finite test data which implies general correctness. Test data of this type

is known as "adequate test data." And, since adequate test data cannot in general be derived algorithmically[4], program testing is not deductive. For this inductive process, we are therefore trying to answer a fundamental question, "If a program is correct on some finite number of test cases, is it correct in general?" Several methods have emerged which allow one to gain confidence in test data adequacy. These methods include path analysis[1,2,5,6] and an associated technique, symbolic execution[7,8]. The basic idea of path analysis is to exercise all control paths within a program. Symbolic execution attempts to derive the test data necessary to do this. Test data known to exercise each flowchart path at least once is considered better than test data that does not. It should be apparent that the possibility of faulty analysis of the program is very real[3].

Let us approach the problem of testing from a different viewpoint and assume that experienced programmers write programs which are either correct or are "almost" correct. Stated more formally:

> If a program is not correct, then it is a "mutant"--that is, it differs from a correct program by simple, well understood errors [11].

Errors have been found to be caused by one of three broad categories[12]. First, the specifications may be misunderstood. Second, the specifications may be

implemented incorrectly. These are the so-called "logical errors." Third, the errors may be of a purely clerical nature. The program mutation methodology can lead to the detection of all three error types[16]. Errors in specification are caught by the programmer when he is forced to understand what his program is going through. Without this understanding, he will not be able to successfully develop test cases. Incorrect implementations are found when functions are not performed (e.g., "dead end code") or are bypassed as a result of some interaction of decisions within the program. Finally, the clerical errors are typically presented to the user as "live" mutants at the end of a series of runs. The user must look at each section of code with a live mutant and determine if this mutant is an alternative method of solving the problem (i.e., an equivalent form of the program) or if the mutant is a degenerate form and thus is in error.

CHAPTER III

PIMS USER'S GUIDE

The remaining sections of this document, in conjunction with the appendices, describes how to use a terminal to operate PIMS on the PRIME-400 computer. All communications to PIMS must be in capital letters. Lower case letters are treated as errors by PIMS.

PIMS consists of three sequentially executed phases which are called the "Pre-Run Phase," the "Mutation Phase," and the "Post-Run Phase." Throughout these phases, errors may occur; and the types of errors detected by PIMS, the error messages, and PIMS' reaction to errors are described in Appendix A. In this chapter it is assumed that no errors take place.

During the Pre-Run Phase, the user tells PIMS which program is to undergo mutation analysis, describes those aspects of the program and the test data needed by PIMS to execute mutations, describes the types of mutations he wants done, and partially describes the contents of his output report. The user may also request that certain status information be displayed on the terminal. During the Pre-Run Phase the user may terminate his run, leaving his transient files unchanged, by issuing a KILL response as a reply to any PIMS prompt for input.

In the Mutation Phase, PIMS creates and executes

mutants. There is no user interaction during this phase.

In the Post-Run Phase, the user completes his description of the output report. He may also request that certain status information be displayed on the terminal.

In the explanations and examples that follow, "space" characters are significant and should be used exactly as shown. In addition, any response to a terminal question should be terminated by a "carriage-return" or "newline" character as appropriate for the user's specific terminal.

The user begins the mutation analysis of a program by converting the original FORTRAN "program" into a subroutine which is valid in the language subset. This subroutine then becomes a Raw Program File. Raw Program Files are created using a text editor prior to entering PIMS. A short tutorial on using this editor is available as a login option (see also Appendix E).

Language Subset Overview

The subset of the FORTRAN language chosen for this implementation of PIMS is such that only INTEGER processing of numeric data is possible. A program must be a SUBROUTINE subprogram with an optional parameter list. Parameters and other variables must be declared using INTEGER or DIMENSION declarations. Arrays may be either one or two dimensional and may be specified in the INTEGER statement.

The acceptable control structures include the logical-IF, GOTO, nested-DO, CONTINUE, and RETURN. Arithmetic expressions may include any of the operators: +, -, *, / or **. Logical expressions are restricted to the IF statement and must be one of: .AND., .OR., or .NOT. as used in many FORTRAN systems. Numeric values should be kept within the range -32,766 to +32,766 due to the nature of the PRIME-400 sixteen bit architecture.

Running PIMS

To begin the execution of PIMS the user types the following command:

OK, SEG RUN>PIMS (see note)

PIMS responds, as soon as it has been loaded, by displaying the following message

PRE-RUN PHASE

ALL INPUT MUST BE IN UPPER CASE

NOTE:
Non-casual users of PIMS should consult with the PIMS staff for details about login and file integrity.

The Pre-Run Phase

The Pre-Run Phase consists of six sequentially executed parts, some optional depending on whether or not PIMS is being run for the first time on the given program. PIMS requests the name of the Raw Program File by displaying the following message:

ENTER THE RAW PROGRAM FILE NAME.

The user types, on the next line, exactly six characters which tell PIMS the file in which the raw program resides. This also sets the file nomenclature convention.

PIMS File-Name Convention

The raw program file name is exactly six characters. We represent this six character name by the symbol <name>. Then the PIMS system files are created and may be accessed with the following suffixes.

<name>.I.....Internal Form File
<name>.T.....Test Data File
<name>.C.....Correctness Descriptor File
<name>.M.....Mutant Information File
<name>.O.....Report Output File
<name>.D.....New Test Data File
<name>.N.....New Mutant Information File
<name>.P.....Predicate-subroutine internal file

NOTE: The user is referred to Appendix E for the details of creating, editing, and maintaining the raw program files which will be processed by PIMS.

PIMS determines the run type, either an initial run

on the program or a subsequent run, by searching for a file with the six character name entered and a suffix of ".I" If this file is found, the run is considered to be a subsequent run; if it is not found, the run is considered initial. Once the system determines that a run is subsequent, the user is given the opportunity to discard all previous files and start over.

During an initial run, PIMS accepts instructions about the routine being tested and any associated test cases. These instructions consist of the sub-parts as described below.

(1) Classification of the Formal Parameters

PIMS requests that the user categorize each formal parameter (for illustrative purposes let the variable be named X) by successively displaying, until all parameters are categorized, the following message:

CATEGORIZE FORMAL PARAMETER x

The user then types the keyword corresponding to one of the categories INPUT, OUTPUT, or INPUT/OUTPUT, or types HELP if he has forgotten details and wants PIMS to display the command keywords.

(2) Mutant Correctness Option

To determine whether mutant correctness is determined by the "predicate subroutine" method or the "same as the program" method, PIMS displays the following message:

> IS MUTANT CORRECTNESS DEPENDENT ON A PREDICATE SUBROUTINE?
>
> TYPE YES OR NO

The user types in the appropriate reply. If YES is entered, PIMS displays the predicate subroutine statement it has found in the Predicate Subroutine File. The user creates this file prior to any initial runs with the appropriate file name as described under file name conventions.

> PREDICATE SUBROUTINE STATEMENT
> (the predicate subroutine name and formal parameters)

(3) Creation of the Test Data File

At this point PIMS is ready to receive the test data from the user and signifies this by displaying the following message:

> HOW MANY TEST CASES ARE TO BE SPECIFIED?

The user enters an appropriate count. For each test case, PIMS prompts the user to enter values for the input formal

parameters of the program. First PIMS requests the values of the scalar parameters, then the one dimensional array formal parameters, and finally the two dimensional arrays, all in a manner to be described below. Requests for a specific test case are signalled by PIMS displaying the following message:

SPECIFY TEST CASE i:

The values of the scalars are requested by PIMS, five at a time, until all scalars are satisfied, by iterating the message

ENTER VALUES FOR V1 V2 V3 V4 V5

The user then inputs the numbers. Should the user have a large volume of test data, he may enter the keyword FILE at this point. The system will ask for a file name and read the test case data from that file. Single dimensioned arrays are input one at a time by PIMS requesting values for specific array elements until all values have been entered. For example, let the array be named A and its dimension be 7. (NOTE: PIMS gets this dimension from the program's DIMENSION statement--it must be either a constant or an input formal parameter scalar variable.) The session would be as follows: PIMS displays

ENTER VALUES FOR A(1) A(2) A(3) A(4) A(5)

The user would enter five numbers. PIMS then displays

```
ENTER VALUES FOR A(6) A(7)
```

The user would enter the final two numbers. PIMS then repeats this process for another one dimensional array or goes on to request values for the two dimensional arrays.

In the case where a user wants to input a partially defined array, he enters UND for the undefined array elements. Only numeric data of the type INTEGER may be processed. PIMS requests the values for two dimensioned array elements in a manner similar to that for single dimensioned arrays. The values are requested in row-major order, five at a time. for example, if A is of dimension (2,7) PIMS will make the following four prompts

```
ENTER VALUES FOR A(1,1) A(1,2) A(1,3) A(1,4)
A(1,5)
ENTER VALUES FOR A(1,6) A(1,7)
ENTER VALUES FOR A(2,1) A(2,2) A(2,3) A(2,4)
A(2,5)
ENTER VALUES FOR A(2,6) A(2,7)
```

(4) Additional Test Cases

When additional test cases can be added to the test case file of the given program, PIMS displays the following message:

HOW MANY NEW TEST CASES FOR THIS RUN?

The user then enters the appropriate count. PIMS then prompts the user to specify the new test cases in the same manner as described above in the "Creation of the Test Data File" subsection. The result is to extend the Test Data File. Test cases cannot be deleted.

(5) Addition of and Status of Mutant Types Considered

To see if new types of mutants are to be considered for this run, PIMS displays the following message:

WHAT NEW TYPES OF MUTANTS ARE TO BE CONSIDERED:

At this point the user has several options. He may type in any of the following replies

NONE - Part (5) terminates.

HELP ----- PIMS displays all the code names of the mutant types as described in Appendix C. Part (5) is then re-executed by PIMS.

ALL ----- Every type of mutation will be considered. Part (5) terminates.

T1 T2 ... Tn ----- T1 ... Tn are code names of mutant types (see Appendix C). Mutants of the listed types will be considered for this and

subsequent PIMS runs. Part (5) is then re-executed by PIMS.

SENSE T1 T2 ... Tn ----- PIMS displays which of the listed mutant types are currently being considered and which are not. Part (5) is then re-executed by PIMS

SENSE ----- PIMS displays which of the possible mutant types are currently being considered and which are not. Part (5) is then re-executed by PIMS.

SELECT T1 T2 ... Tn ----- The user specifies to PIMS which of the mutants he wishes to know about.

(6) Display and Output of Past Results

In order to inform the user, in non-initial runs, that part (6) has begun, PIMS displays the following message:

REVIEW PREVIOUS RUN RESULTS

At this point the user has three options: (1) he may request that certain information concerning the mutant status before this run be displayed, (2) he may request that similar information be included in his output file, or (3) he may request that the mutation phase of PIMS be started.

Option three is requested by typing MUTATE. The other two options cause this display to be re-executed, accomplished by a repeated displaying of the "REVIEW..." message. The information which can be displayed or included in the output file is the following:

(a) Displaying Information

All requests to display information on the screen begin with the word DISPLAY. Next there is a space followed by a keyword which describes the information to be put on the screen. The keywords are the following:

HEADER --- The program subroutine statement and the classification of the program's formal parameters are displayed.

CORRECTNESS --- The method of determining mutant correctness and, possibly, the subroutine statement of the predicate subroutine are displayed.

TITLE --- The PIMS run title is displayed.

STATUS --- The mutants' status before this run is displayed

(b) Outputting Information

All requests to have information included in the

user's output file begin with the word OUTPUT. Next there is a space followed by a keyword describing the type of information to be included in the output file. The keyword is the following:

TESTCASES --- The previous test cases are included.

TESTCASE n --- The specified case, "n" is displayed.

The Mutation Phase

There is no user interaction during the mutation phase. In the event of a fatal processing error, the host operating system will issue appropriate diagnostic messages.

The Post-run Phase

The Post-Run Phase consists of one part which is similar to part (6) (Display and Output of Past Results) of the Pre-Run Phase. It can be called "Display and Output of New Results." In order to inform the user that the Post-Run Phase has begun, PIMS displays the following message:

POST-RUN PHASE

At this point the user has three options: he may request that certain information concerning the mutant results for

this run and the mutant status after this run be displayed, he may request that similar information as well as mutant program listings be included in his Output file, or he may request that the PIMS run terminate. The first two options will be described below. The third option is requested by typing STOP and in each of the former two options the Post-Run Phase re-cycles by PIMS displaying the following message:

POST RUN RESULTS

The information which can be displayed or included in the output file is the following:

(1) Display of Information

All requests to display information on the screen begin with the word DISPLAY. Next there is a space followed by a keyword describing the information to be put on the screen. The keywords are the following:

(i) HEADER - Same as in the Pre-Run Phase.

(ii) CORRECTNESS - Same as in the Pre-Run Phase.

(iii) TITLE - The PIMS run title is displayed.

(iv) RESULTS - The mutant results for this run are displayed.

(v) STATUS - The mutants' status after this run is

displayed.

(2) Output of Information

All requests to have information included in the user's output file begin with the word OUTPUT. Next there is a space followed by a keyword describing the type of information to be included in the output file. The keywords are the following:

(i) TESTCASES - The new test cases are included.

(ii) MUTANTS - This keyword must be followed by additional keywords as follows: (The absence of keywords implies the ALL keyword.)

(a) ALL - A listing of all the live mutants is included.

(b) RANDOM - A listing of one randomly selected live mutant of each possible mutant type having live mutants is included.

(c) ALL T1 T2 ... Tn - Where T1 through Tn are the keywords for each desired mutant type. A listing of all the live mutants for each of the given types is included.

(d) RANDOM T1 T2 ... Tn - A listing of one randomly selected live mutant for each of the given mutant types is included.

(e) HELP - PIMS displays the code names of the mutant types as described in Appendix D.

Once a user decides to list a mutant type, via either an ALL or a RANDOM, he cannot later switch to no listing for the type. However, he may switch from ALL to RANDOM or from RANDOM to ALL for any mutant type. The system will not produce a listing of any mutant types until explicitely told to do so.

CHAPTER IV

IMPLEMENTATION AND PORTABILITY DISCUSSIONS

Implementation

The PIMS program is written in FORTRAN as a feasibility study of automatic program mutations. In other words, we addressed the question: "Can the concept of program mutation be implemented in an automated system with reasonable runtime and computational simplicity?" The top levels are depicted in the following diagram.

```
Driver-----PRERUN-----CLRTTY,CIFILE,CCFILE,CTFILE,CMFILE,
          |            DISPLY,FILEOP,GETNAM,GETTYP,LTINFO,
          |            MMRECS,NEWTST,WPAST,WGSTAT
          |
          --MPHASE-----CLRTTY,DISPLY,MERGED,MERGEM,XNEWMU,
          |            XOLDMU
          |
          --POSTRN-----CLRTTY,CLENUP,DISPLY,NEWRES,WNEW
```

Beyond these levels, the control paths are relatively difficult to analyze from a maintenance programmer's point of view. The system data structures are almost entirely parametric and allocation is contained in multiple COMMON blocks. The large number of these blocks and their extensive use permits many side-effects to take place as the result of procedures invoked at every level. These side effects also greatly complicate the issues concerning the scope-of-control of procedures over their variables.

PIMS executes in a paged environment as a single image with about 67K bytes of address space required for both the data and the executable code. Since the program is logically divided into three distinct phases, the address space could be reduced with little impact on execution or operation by implementating the task as overlays or separately executed programs. This approach is recommended for implementing the PIMS program on most minicomputers.

Portability

During the implementation of PIMS on the PRIME-400, much collaboration took place between the research groups at Yale University and at Georgia Tech. My efforts used a sixteen-bit machine, the PRIME-400, while the Yale effort used a 36-bit machine, the DECsystem-10. The only medium available for transporting programs and data between these two systems was nine-track magnetic tape.

Although both vendors claimed to support ANSI compatible magnetic tapes, files could not be written by one system and read by the other without some form of intermediate processing. A list of this processing includes:

1) Records which were written with 80 characters per record and one record per block used different methods

of indicating the end-of-record. Specifically, DEC wrote an 82-character record with two trailing nulls (binary zeroes) while PRIME expected an 80-character record which included the two nulls.

2) Tape files with embedded carriage-return/line-feed sequences caused general havoc on both systems. The line-feeds usually had to be removed before any progress was made in processing the files.

A second and larger set of problems was encountered when FORTRAN source files were moved between the two systems. Obvious problems developed as a result of the differing word lengths and associated integer magnitude. The impact of many of these problems was lessened by the PRIME FORTRAN declaration for long-integers, INTEGER*4, which specifies a 32-bit integer. In order to perform the same functions on divers computers, it would be necessary to constrain all implementations. A discussion of those constraints is presented below.

First, all integer quantities should be kept within the range -32,767 to +32,767. This would allow the system to function on sixteen-bit machines that do not provide any long integer forms.

Second, the packing of multiple fields of data per integer variable should be avoided. This packing is also

inefficient for the large word machines, but there are severe unpacking problems for the sixteen-bit machines. In our case, a 36-bit word was used on the DECsystem-10 to contain two nine-bit and one eighteen-bit fields. We were able to implement this using a long integer and obtain two eight-bit and one sixteen-bit fields. This loss of magnitude has not caused problems to date. Expanded range can be obtained by segmenting the use of the system to process smaller programs.

Third, the character processing that is done in the compiler and command processor should be done either with integer tokens subject to the first constraint. If integer tokens are not desirable, then at least characters should be processed in FORTRAN A1 format. The A1 processing will decrease the efficiency for the large word machines again, but the character routines will be portable. A good machine independent "string subroutine" package would probably be a better choice here.

Fourth, vendor supplied features and all I/O should be imbedded in user written procedures. In some cases, this will merely add a layer of run-time linkage with the parameters to the user routine being passed directly to the vendor feature or I/O routine. However, this layer allows other system routines to be substituted and code added to provided for the behavior of another technique. Modifying a single imbedded routine is much easier than searching for

all of the uses of a specific statement throughout an entire system implementation.

We believe that the above techniques should be used in future implementation efforts. They were not used in our system but the benefits of these techniques became apparent as we tried to pass more and more FORTRAN source data between machines.

CHAPTER V

SUMMARY

A program mutation system was built and is now operational on a PRIME-400 minicomputer. The user may specify an input data file that contains a subroutine which is valid in a certain subset of the FORTRAN programming language. This subroutine is parsed, interpreted with user specified test data and the user is given the opportunity of determining the correctness of this test data either manually or through the use of a predicate subroutine that will determine the correctness of this base routine.

Once the user thinks he has an adequate test data set, this base program is modified in several ways and executed again after each modification. These modifications are called mutants and each mutant will either survive or die during its execution. All mutants that produce incorrect results or will not be valid subset programs will die. Those mutants that produce correct results will indicate to the user that further analysis is needed.

When further analysis is necessary the user must determine that either a live mutant is equivalent with other mutants and discard the equivalent mutants manually, or a live mutant might be eliminated by augmenting the test data set. The test data set is then modified as required and the mutants that remain live are executed again. Each time, the

program will report various statistics about the live mutants remaining. When the user is satisfied with completeness of data achieved, the process stops. We now say that an acceptable level of test data adequacy has been reached. In more quantitative terms, some percentage of the total mutants will remain live at this point. For tests of the same subroutine, we interpret this percentage to mean, the test data that shows the lowest number of live mutants at the time of comparison, has the most adequate test data. We use this measure of test data adequacy to infer that the subroutine with the more adequate test data has been tested more thoroughly. In addition, the subroutine that has been tested more thoroughly is more probably the most correct.[20]

Observations on PIMS Utilization

Three programs were tested on PIMS using mutation analysis. The results of these tests are described in detail in [15]. The three programs were: FIND, by Hoare; PAT, by Knuth, Morris, and Pratt; and SCAN from PIMS' lexical analyzer. The following testing strategy was adopted for the testing of these programs. "Good test data" was constructed for each program independently of the mutation system. This data and the program source were input to the mutation system with all mutation operators in

effect. The results of the testing for SCAN will be described here.

In the initial PIMS run, SCAN presented 104 executable statements and 19 test cases were constructed. When all mutants had been generated, there were 8,838 mutants created of which 89.1% failed. In this sense, failure means: produced incorrect or syntactically invalid programs; caused a processing abort due to time or error condition; or, produced inaccurate results. The original data set was augmented until all mutants either failed some test or were determined to be equivalent to the original program.

RESULTS OF TESTING "SCAN" WITH PIMS

Executable statements:	104
Mutants Created:	8838

	Test Cases	% Eliminated
Initial Run	19	89.1
Final Run	35	97.9

An analysis of the initial and final elimination percentages shows that our original data set, which was manually-generated, "good test data," failed to distinguish 778 incorrect mutants in SCAN. In addition, experiments run

on other programs seem to suggest that two percent is a good estimate for the expected number of equivalent mutants.[15]

When all mutants had been eliminated as described above, 35 test cases had been constructed and submitted over a course of seven iterations of the PIMS mutation process. At the end of this procedure, 97.9% of the mutants had failed. When the results of the first and last mutation run are compared, one reason why program testing is held in low esteem becomes apparant: **"Even after hard thought, the initial test data failed to distinguish a large number of incorrect mutants."[15]**

At this point a word about the operating environment is in order. The analysis of SCAN described above was performed on a Digital Equipment Corp. PDP-10 processor. The unclever pilot mutation system required 90 minutes of CPU time to interpret the 8,838 mutants of the 104 statement scanner. Since the PDP-10 used is five times slower than an IBM 370/158 and thirty times slower than a CDC-7600, and since our task is processor bound, the 90 minute execution time scales down directly to eighteen minutes on the IBM machine and three minutes on the CDC processor. On a reasonably fast processor, these appear to be acceptably cost-effective execution times.[15]

Conclusion

Program Mutation is a valuable asset in program testing. The methodology greatly reduces the time and effort required to find errors in those programs studied thus far. Although there is a wealth of FORTRAN software in the world today, it is difficult to obtain and modify real-world software for anaylsis. This problem is the topic of current research.

A consideration which must be taken into account is the order in which the alternatives or mutants are introduced into the system. One viewpoint is that each mutant should be considered in total isolation from all the others. This has the advantage of conceptual simplicity, but the number of test cases thereby generated is unnecessarily large.

Making the not unwarranted assumption that the time required for a test case generation is significantly larger than that required for a mutant execution, the PIMS methodology appears needlessly complex. It has been observed in almost all testing methods that there is a strong degree of collateral testing. That is, test cases which were designed to eliminate one error will often also have the effect of eliminating other related errors. This collateral elimination appears to hold true for PIMS as well. However, there may be some difficulty in recognizing

this collateral relationship solely from the descriptions of the mutants involved.

As a result of collateral elimination, an alternative algorithm for mutant elimination might be formulated as:

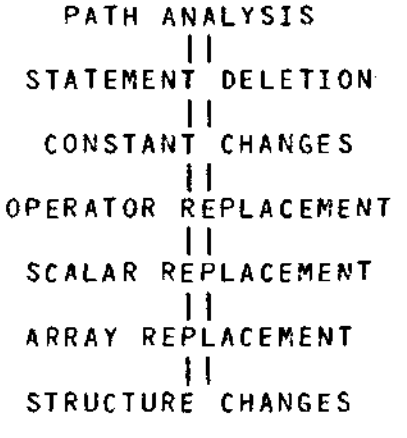
(1) Choose any one mutant

(2) Generate a test case to eliminate the mutant.

(3) Run all remaining mutants against the new data.

This new methodology would result in a reduction in the number of mutants which are actually used to drive the test case generation process.

As yet, the order of mutation selection remains unspecified. Consider that if a statement is never executed, it may be altered in any way and no test data will detect the changes. One hierarchy for mutant selection has been proposed by Tim Budd, University of California, Berkly. This hierarchy is shown:



The table shown below lists the relative frequency of errors found in a sample of FORTRAN, COBOL, PL/1, and BASIC programs. Other types of errors were reported, most notably I/O and data errors, and the relativer frequencies were all approximately 0.05. However, the table shows errors which are currently detected by PIMS. It should be noted that the frequencies for the first three items are high, in specific, the first three items shown are three of the highest six error types in the entire sampling. Since PIMS successfully detects five of the error types with a frequency of 0.05 or better, we feel that there is a significant contribution to be made by this technique.[13]

ERROR TYPE BY FREQUENCY

Error in assignment or computation	.27
Unsuccessful iteration	.09
Branching error	
Unconditional	.01
Conditional	.05
Subscript Violation	.05
Subprogram Invocation Error	.05
Failure to Terminate	.01

The PIMS system discussed here was not designed with the properties of portability and maintainability. I would like to suggest that these topics are suitable for investigation in their own right. The idea of program mutation is currently being extended to full ANSI-1969 FORTRAN, COBOL, and PASCAL with the hope that by examining the effects of the methodology in several programming

languages, some insight may be obtained into the methodologies of program testing in general, and program mutation in specific.

APPENDIX A

ERROR MESSAGES

This document describes the errors detected by PIMS during the interactive phases of a PIMS run, the messages displayed by PIMS on detection of an error, and the actions taken by PIMS after finding an error. The errors are divided into two classes: fatal and non-fatal, with fatal errors resulting in an abort of the PIMS run. Fatal errors only occur during the Pre-Run Phase of PIMS. The occurence of a fatal error or the user entering KILL during the Pre-Run Phase of PIMS leaves all transient files as they were before the PIMS run began. Once the user issues a GO command, thus signalling the end of the Pre-Run Phase, he will not be able to issue a KILL.

We also group the errors into those which occur during parsing the program and those which occur strictly as bad responses to prompts made by PIMS. Parser errors always are fatal and the PIMS parser is designed to abort on detection of a first error. That is, if the program has multiple syntactic errors, the PIMS parser will print an error message for only the first of them.

Parser Error Messages

The messages which the user may encounter during the parsing of either the routine which is being tested or of the predicate subroutine are very similar to those generated by any FORTRAN compiler. Since a knowledge of FORTRAN is prerequisite to a meaningful use of the PIMS system, an understanding of typical compiler diagnostics is assumed.

One aspect of the PIMS diagnostics that is different from that of a typical compiler is the fatal nature of compiler diagnostics. In the event that any compile-time error is encountered within a module, the error is reported at that point and the compile is aborted. There is no attempt at compile-time error trace-back or recovery. If any errors are reported, the user is advised to scan the remainder of the routine manually for other syntax errors prior to a resubmission to PIMS. This manual scan should save man and machine time during the Pre-run Phase.

Interactive Error Messages

Pre-Run Phase

(a) The Program Name

(1) Message: ILLEGAL FILE NAME
Action: Repeat part (a).

(2) Message: NON-EXISTENT RAW PROGRAM FILE
Action: Repeat part (a).

(3) Message: FILE NAME CONFLICT - OUTPUT FILE ALREADY EXISTS
Action: None. Serves as a warning.

(b) The Run Type

(1) Message: ILLEGAL REPLY
Action: Repeat part (b).

(2) Message: PROGRAM NOT IN THE PIMS FORTRAN SUBSET
Action: Abort

(3) Message: THE FOLLOWING TRANSIENT FILES ARE MISSING:
Action: Abort

(4) Message: THE FOLLOWING TRANSIENT FILES ALREADY EXIST:
Action: Abort

(c) Program and Test Cases

(1) Message: ILLEGAL CLASSIFICATION
Action: (A) Display the legal classification codes.
(B) Repeat part (c-1) on the same parameter.

(2) Message: ILLEGAL REPLY
Action: Repeat Program and Test Cases

(3) Message: PREDICATE SUBROUTINE FILE DOES NOT EXIST
Action: Abort

(4) Message: BAD PREDICATE SUBROUTINE CALLING SEQUENCE
Action: (A) Display the program's formal parameters and their classifications.
(B) Display the predicate subroutine statement.
(C) Abort

(5) Message: ILLEGAL VALUE
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(6) Message: NOT ENOUGH DATA SUPPLIED
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(d) Additional Test Cases

(1) Message: ILLEGAL VALUE
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(2) Message: NOT ENOUGH DATA SUPPLIED
Action: Repeat the request for data on the same input formal parameter(s). User's input ignored.

(e) Addition of and Status of Mutant Types Considered

(1) Message: ILLEGAL REPLY
Action: (A) Display all legal replies.
(B) Repeat part (e).

(2) Message: ILLEGAL MUTANT TYPE
Action: (A) Display the coded names of the mutant types.
(B) Repeat part (e).

(3) Message: THESE MUTANT TYPES WERE ALREADY ON:
Action: None. Serves as a warning. The other

specified mutant types which were off are now on.

(f) Displaying and Outputting of Past Results

(1) Message: ILLEGAL REQUEST
Action: (A) Display the legal requests for part (f).
(B) Repeat part (f).

(g) General Errors

(1) Message: PROGRAM FAILS
Action: (A) Display the test case on which it fails.
(B) Display the way in which it failed.
(C) Put (A) and (B) in the output file.
(D) Abort

(2) Message: PREDICATE SUBROUTINE FAILS
Action: (A) Display the test case on which it fails.
(B) Display the way in which it fails.
(C) Put (A) and (B) in the output file.
(D) Abort

(3) Message: OUTPUT FILE EXISTS - TYPE KILL OR CONTINUE.
Action: Abort on KILL, delete output file on CONTINUE.

(4) Message: TOO MUCH DATA OR FAULTY SYNTAX
Action: Repeat the previous prompt for numeric input.

(5) Message: ILLEGAL VALUE
Action: Repeat the previous prompt for numeric input.

The Post-Run Phase

(a) Message: ILLEGAL REQUEST
Action: (A) Display the legal requests for the Post-Run Phase.
(B) Repeat the Post-Run Phase.

(b) Message: ILLEGAL MUTANT TYPE.
Action: (A) Display all legal mutant types.
(B) Repeat the Post-Run Phase.

APPENDIX B

FORTRAN LANGUAGE SUBSET

This appendix describes the FORTRAN subset language whose programs can be tested using the Pilot Mutation System. Only the syntax of this subset, specified in an extended BNF (see below), is given. The syntax presented is in a "pure" form with the mundane aspects of FORTRAN syntax assumed. These include the following: 1) statements start on a new line and appear in "card" columns 7-72, 2) column 6 is the statement continuation column, 3) statement labels appear in columns 1-5, 4) names have lengths of 6 or less characters, and 5) comment statements have a C in column 1.

The PIMS FORTRAN subset has the following two semantical restrictions: (1) all variables must be declared, and (2) keywords, such as DO and END, cannot be used as variable names.

BNF Description of the Language Subset

Standard BNF is augmented with the following four abbreviations:

(1) list appendix - <y> ::= <x-list> is equivalent to

<y> ::= <x> | <x> <y>

(2) commalist appendix - <y> ::= <x-commalist> is equivalent to

<y> ::= <x> | <x> , <y>

(3) option - <y> ::= <x> [<z>] is equivalent to

<y> ::= <x> | <x> <z>

(4) choice - <y> ::= <x> {<w> | <z>} is equivalent to

<y> ::= <x> <w> | <x> <z>

Programs

<program> ::= SUBROUTINE <program-name> (
<formal-argument-commalist>)
  <declaration-statement-list>
  <executable-statement-list>
  END

Formal Arguments

<formal-argument> ::= <variable-name>

Declaration Statements

<declaration-statement> ::= INTEGER <declaration-commalist>

<declaration> ::= <scalar-decl> | <array-decl>

<scalar-decl> ::= <variable-name>

<array-decl> ::= <one-dim-array-decl> | <two-dim-array-decl>

<one-dim-array-decl> ::= <variable-name> (<limit>)

<limit> ::= <positive-integer> | <variable-name>

<two-dim-array-decl> ::= <variable-name> (<limit-pair>)

<limit-pair> ::= <limit> , <limit>

Executable Statements

<executable-statement> ::= [<label>] <statement>

<label> ::= <positive-integer>

<statement> ::= <simple-statement> | <conditional-statement>
| <do-loop-statement>

Simple Statements

<simple-statement> ::= <goto-statement> |
<assignment-statement>
<continue-statement> | <return-statement>

<goto-statement> ::= {GO TO | GOTO} <label>

<assignment-statement> ::= <reference> =
<arithmetic-expression>

<continue-statement> ::= CONTINUE

<return-statement> ::= RETURN

Conditional Statements

<conditional-statement> ::= IF (<logical-expression>)
<simple-statement>

DO-loop Statements

<do-loop-statement> ::= <index-part>
  <outer-loop-body>
  <loop-end>

<index-part> ::= DO <label> <index> = <initial> , <terminal>
[, <increment>]

<outer-loop-body> ::= <outer-loop-statement-list>

<outer-loop-statement> ::= [<label>]
  {<simple-statement> |
  <conditional-statement> |
  <inner-do-loop> }

<inner-do-loop> ::= <index-part>
  <loop-body>
  [<loop-end>]

<loop-body> ::= <loop-statement-list>

<loop-statement> ::= [<label>]
  {<simple-statement> | <conditional-statement>}

<loop-end> ::= <label> <loop-end-statement>

<loop-end-statement> ::= <continue-statement> |
  <assignment-statement> |
  <conditional-statement>

<index> ::= <scalar-reference>

<initial> ::= <scalar-reference> | <positive-integer>

<terminal> ::= <scalar-reference> | <positive-integer>

<increment> ::= <scalar-reference> | <positive-integer>

Arithmetic Expressions

<arithmetic-expression> ::= [<arithmetic-expression> {+ |
-}] <ae3>

<ae3> ::= [<ae3> {* | /}] <ae2>

<ae2> ::= [<ae2> **] <ae1>

<ae1> ::= <primitive-ae> | - <ae1> | (
<arithmetic-expression>)

<primitive-ae> ::= <reference> | <integer>

Logical Expressions

<logical-expression> ::= [<logical-expression> .OR.] <le2>

<le2> ::= [<le2> .AND.] <le1>

<le1> ::= <primitive-le> | .NOT. <le1> | (

<logical-expression>)

<primitive-le> ::= <arithmetic-expression> <relational-op>
<arithmetic-expression>

<relational-op> ::= .LT. | .LE. | .EQ. | .NE. | .GT. |
.GE.

Data References

<reference> ::= <scalar-reference> | <array-one-reference> |
<array-two-reference>

<scalar-reference> ::= <variable-name>

<array-one-reference> ::= <variable-name> (<simple-ae>)

<array-two-reference> ::= <variable-name> (<simple-ae> ,
<simple-ae>)

<simple-ae> ::= [-] [<positive-integer> *]
<scalar-reference> {+ | -}
<positive-integer> |
[-] <scalar-reference> |
<positive-integer>

Identifier Names

<program-name> ::= <name>

<variable-name> ::= <name>

<name> ::= <letter> [<alphameric-list>]

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L |
M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<alphameric> ::= <letter> | <digit>

<digit> ::= <zero> | <positive-digit>

<zero> ::= 0

<positive-digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Constants

<constant> ::= <integer>

<integer> ::= <positive-integer> | <zero-list> | -
<positive-integer>

<positive-integer> ::= <positive-digit> [<digit-list>]

APPENDIX C

COMMANDS AND ABBREVIATIONS

This appendix describes the commands and their abbreviations that are used to communicate with PIMS during the interactive phases. The commands for specifying mutant types follow.

The user specifies mutant types to PIMS by using the following three character abbreviations. An abbreviation marked * means the mutant type is not currently implemented. There are no "full word" commands for specifying mutant types.

(a) Data Declaration Mutations

(i) ALD - Array Limit Default Insertion

(ii) ALP - Two Dimensional Array Limit Permutation

(b) Data Reference Mutations

(i) CRP k - Constant Replacement. The value k>=1 gives the neighborhood (i.e., c +/- k) of the replacing constants. The user may choose not to specify k, in which case a default value of k=1 is assumed (see Appendix D).

(ii) SVR - Scalar Variable Replacement

(iii) SFC - Scalar Variable for Constant Replacements

(iv) CFS - Constant for Scalar Variable Replacement

(v) CAR - Comparable Array Name Replacement

(vi) CFA - Constant for Array Reference Replacement

(vii) SFA - Scalar Variable for Array Reference Replacement

(viii) AFC - Array Reference for Constant Replacement

(ix) AFS - Array Reference for Scalar Variable Replacement

(x) AIP - Two Dimensional Array Index Permutation

*(xi) SVI k - Scalar Variable Initialization Insertion. The value >0 gives the i+k set of initializing values. The user may choose not to specify k, in which case a default value of k=0 is assumed.

(c) Operator Evaluation Mutations

(i) AOR - Arithmetic Operator Replacement

(ii) ROR - Relational Operator Replacement

(iii) LCR - Logical Connector Replacement

*(iv) APP - Arithmetic Precedence Permutation

*(v) LPP - Logical Precedence Permutation

(d) Control Mutations

(i) GLR - Goto Label Replacement

(ii) PAN - Path Analysis

(iii) CSI - Continue Statement Insertion

(iv) CSD - Continue Statement Deletion

*(v) ILD - Inner Do Loop Decoupling

*(vi) DIA - Do Loop Index Alteration

(vii) RSR - Return Statement Replacement

APPENDIX D

DESCRIPTION OF THE MUTATIONS PERFORMED

This appendix describes the types of first order mutations which the Pilot Mutation System considers and some other mutations, marked with a "*", which may be considered in future extensions of PIMS. The wording used is tied to the syntactic categories defined in the "FORTRAN Language Subset" (see Appendix B). Only those programs which are in the subset language are considered to be valid mutations.

Data Declaration Mutations

Array Limit Default Insertion is accomplished by replacing each scalar reference in an array declaration by 1.

Two Dimensional Array Limit Permutation is accomplished by exchanging each two dimensional array declaration limit pair.

Data Reference Mutations

The following sets are referenced in defining the mutation operations in this section:

A --- set of all array references appearing in the program.

C --- set of all constants appearing in executable statements of the program.

K --- the set {-k,-k+1,...,-1,0,1,...,k-1,k} where k> 0 is supplied by the user

S --- set of all scalar variable names appearing in executable statements of the program

V1 --- set of all one dimensional array names appearing in executable statements of the program.

V2 --- set of all two dimensional array names appearing in executable statements of the program.

1. Constant Replacement

Each constant c appearing in any executable statement is replaced by members of the set {c-k,c-k+1,...,c-1,c+1,...,c+k}. If k=0 is supplied, then no constant replacements are produced by PIMS.

2. Scalar Variable Replacement

Each scalar variable s appearing in any executable statement is replaced by members of S-{s} .

3. Scalar Variable for Constant Replacement

Each constant c appearing in any executable statement is replaced by members of S

4. Constant for Scalar Variable Replacement

Each scalar variable s appearing in any executable statement is replaced by members of C.

5. Comparible Array Name Replacemment

Each instance of v1 in V1 appearing in any executable statement is replaced by members of V1-{v1}. Each instance of v2 in V2 appearing in any executable statement is replaced by members of V2-{v2}.

6. Constant for Array Reference Replacement

Each instance of ar in A appearing in an executable statement is replaced by members of C.

7. Scalar Variable for Array Reference Replacement

Each instance of ar in A appearing in an exectable statement is replaced by members of S.

8. Array Reference for Constant Replacemment

Each instance of c in C appearing in any executable statement is replaced by members of A

9. Array Reference for Scalar Variable Replacement

Each instance of s in S appearing in any executable statement is replaced by members of A

10. Two Dimensional Array Index Permutation

Each instance of references to two dimensional arrays has its indecies permuted

*11. Scalar Variable Initialization Insertion

For each s in S the initial value of s is set to members of K.

Operator Evaluation Mutations

1. Arithmetic Operator Replacement

Each instance of a binary operator bo is replaced by members of the set {+,-,*,/,**}-{bo}. Each instance of unary - is eliminated.

2. Relational Operator Replacement

Each instance of a relational operator ro is replaced by members of the set {.LT.,.LE.,.EQ.,.NE.,.GT.,.GE.}-{ro}.

3. Logical Connector Replacement

Each instance of .AND. is replaced by .OR., each instance of .OR. is replaced by .AND., and each instance of .NOT. is eliminated.

*4. Arithmetic Precedence Permutation

Each arithmetic expression containing>1 arithmetic operators

is replaced by each of its distinct alternative parses.

*5. Logical Precedence Permutation

Each logical expression containing>1 logical connectors is replaced by each of its distinct alternative parses.

Control Mutations

The following sets of statement labels are referenced in defining the mutation operations in this section:

L --- set of all statement labels in the program.

TRAP --- used to represent a statement which is guaranteed to cause a program interrupt

1. Goto Label Replacement

Each instance of l in L in any goto statement is replaced by members of L-{l}.

2. Path Analysis

Each simple statement (including those which are imbedded in conditionals) and each conditional statement is replaced by TRAP. Each index part of each do loop statement has a TRAP inserted as its subsequent statement. This checks that each control path is traversed at least once and can easily be extended to see if each path is traversed any number of times.

3. Continue Statement Insertion

Each do loop which does not end on a continue statement is made to do so

4. Continue Statement Deletion

Each do loop which ends on a continue statement is made to end on the preceeding statement.

*5. Inner Do Loop Decoupling

Inner do loops which end on the same statement as their containing do loop are made to end on a separate, possibly duplicated, statement.

*6. Do Loop Index Alteration

Although this type of mutation is not currently implemented as a separate type, these mutations can be produced as a result of data mutations (see above).

7. Return Statement Replacement

Each non-return simple statement (including those which are imbedded in conditionals) and each conditional statement is replaced by a return statement. Each index part of each do loop statement has a return statement inserted as its subsequent statement.

APPENDIX E

ENTERING AND MODIFYING FILES

Programs are normally entered into the computer using the PRIMOS[18] Text Editor (ED). This editor is a line oriented text processor whose line pointer is always located at the last line processed (whether the processing is printing locating, moving pointer, etc.). The Editor operates in one of two modes, INPUT mode or EDIT mode.

When creating a new file, the Editor is invoked by typing

OK, ED

which places the Editor in the INPUT mode. When modifying an existing file, the Editor is invoked by typing

OK, ED filename

which places the Editor in the EDIT mode. The "filename" specified is the six-character name assigned to the raw program file being created or modified. At any time, the user may type a carriage return (c/r) with no other characters preceding it. This is known as a "null response." This null response will switch the Editor from the EDIT mode to the INPUT mode or from INPUT mode to EDIT mode.

INPUT Mode

The INPUT mode is used when entering text information into a file (e.g., creating a program). The word INPUT is displayed at the user's terminal to indicate that the Editor has entered the INPUT mode. The c/r key will terminate the current line of text and prepare the Editor to receive a new line. Tabulation is accomplished with the backslash (\) character. Each backslash represents the first, second, etc. tab setting; the tab stops are at columns 6, 15, and 30. The use of c/r with no text preceding it puts the Editor in EDIT mode.

EDIT Mode

The EDIT mode is used when the contents of a file are to be modified. More than 50 commands are available, although we will only describe a subset of the available commands that should suffice for most purposes. The commands are described later in this appendix.

In the EDIT mode, the Editor maintains an internal line pointer at the current line (the last line processed). The commands TOP, BOTTOM, FIND, and LOCATE, move this pointer. The WHERE command displays the current line number; POINT moves the pointer to a specified line number. The MODE NUMBER command causes the line number to be displayed whenever a line of text is displayed. All

commands for location and modification begin processing with the current line. The use of c/r with no text preceding it puts the Editor in INPUT mode.

Typographical Error Correction

In either mode the user may correct errors in typing before the terminating (c/r) is typed. The last character entered is deleted, moving from right to left, one character for each backspace(b/s) typed. The entire current line may be deleted by typing the delete(del) character. The character (b/s) is obtained by holding the key marked "CTRL" or "CONTROL" and then striking the key "H." Any line followed by the delete character is null, and a (c/r) at that point will switch the editor into the alternate mode.

Saving Files

Orderly termination of an Editor session is done from the EDIT mode. The command:

FILE filename

writes the current version of the edited file to the disk under the specified filename. The file will be created if it did not previously exist or it will be overwritten if it does exist. If an existing file is being modified, the command:

FILE

writes the new version to the disk with the old filename. After the execution of the FILE command, the Editor is terminated and control returns to PRIMOS signified by: "OK," on the user terminal.

Other Useful Techniques

The following general descriptions will aid the user in adapting to the PRIMOS Editor.

Any number of lines may be moved from one location to another using the DUNLOAD command. DUNLOAD deletes these lines as it writes them into an auxiliary file. A LOAD command loads the new auxiliary file data at the desired point. Any number of lines may be copied from one location to another using the UNLOAD command. UNLOAD works the same as DUNLOAD except that UNLOAD does not delete the lines as they are being written.

Any line the begins with a legal FORTRAN statement number may be located with the FIND command.

The MODIFY command is used when a line must be altered but the relative column alignment must remain the same.

EDITOR Command Summary

The following is an alphabetical list of some of the available Editor commands. For a detailed description of all commands, the user is referred to the Editor Reference Section of THE NEW USER'S GUIDE TO EDITOR AND RUNOFF[19]. In the following descriptions, the parameter "string" is any series of ASCII characters including leading, trailing, or embedded blanks.

APPEND string.....Appends string to the end of the current line.

BOTTOM.....Moves the pointer beyond the last line of the file.

CHANGE/st1/st2/[n] [G].....Replaces st1 with st2 for n lines. If G is omitted, only the first occurrence of st1 on each line is changed; if G is present, all occurrences on n lines are changed.

DELETE [n].....Deletes n lines, including the current line. The default value of n is one.

DUNLOAD filename [n].....Deletes n lines from the current file and writes them into filename. The default value of n is one.

FILE [filename].....Writes the contents of the current file into filename and QUITS to PRIMOS.

FIND string.....Moves the pointer to the first line beginning with string.

INSERT string.....Inserts the string after the current line.

LOAD filename.....Loads text from filename into the current file following the current line.

LOCATE string.....Moves the pointer forward to the first line containing string. The string may contain leading and trailing blanks.

MODE NUMBER.....Displays line numbers in front of displayed lines.

MODE NNUMBER.....Turns off the line number

display.

NEXT [{+|-}] [n].....Moves the pointer n lines, forward if n is positive and backward if n is negative.

POINT [n].....Moves the pointer to line n.

PRINT [n].....Displays the current line or n lines beginning with the current line.

QUIT.....Terminates the editing session without filing the current file.

RETYPE string.....The current line is replaced by string.

TOP.....Moves the pointer one line before the first line of text.

UNLOAD filename [n].....Copies n lines into filename.

WHERE.....Displays the current line number.

Other Capabilities Outside The EDITOR

From time to time the user will probably wish to view

the contents of a file, delete an existing file or change the name of an existing file. These capabilities exist outside of the Editor facilities. In order to view a file at the user's terminal, the user types

OK, SLIST filename

where filename is the name of the file to be listed. Upon completion of the listing, control is returned to PRIMOS.

Files may be deleted with the PRIMOS command

OK, DELETE filename

where filename is the name of the file to be deleted. A user may not delete a file that he does not own or that has been appropriately protected.

Files may be renamed with the PRIMOs command

OK, CNAME oldname newname

where oldname is the current name of the file and newname is the desired new file name. A user may not rename a file that he does not own or that has been appropriately protected.

APPENDIX F

SAMPLE PIMS RUN

The following is a copy of the terminal dialog from an initial PIMS run. Some of the lines were changed to fit them on the page, but the information presented is unchanged.

```
OK, SEG RUN>PIMS

PRE-RUN PHASE
ALL INPUT MUST BE IN UPPER CASE

 ENTER THE RAW PROGRAM FILE NAME
JBST02

 CATEGORIZE FORMAL PARAMETER N
PROG
    1          SUBROUTINE SORT02(N,A)
    2   C   *BUBBLE SORT - ALLOW EARLY TERMINATION
    3          INTEGER N,A(N)
    4          INTEGER I
    5          INTEGER T
    6          INTEGER SORTED
    7   C
    8          IF (N.LE.1) GOTO 300
    9      100 CONTINUE
   10          SORTED = 1
   11          DO 200 I = 2,N
   12            IF (A(I-1).LE.A(I)) GOTO 200
   13              T      = A(I-1)
   14              A(I-1)  = A(I)
   15              A(I)= T
   16              SORTED = 0
   17      200 CONTINUE
   18          IF (SORTED.NE.1) GOTO 100
   19      300 CONTINUE
   20          RETURN
   21          END

 TYPE NEXT COMMAND
INPUT
```

```
 CATEGORIZE FORMAL PARAMETER A
IO

 IS MUTANT CORRECTNESS DEPENDENT ON A PREDICATE SUBROUTINE?
 TYPE A YES OR NO    ****
NO

 HOW MANY TEST CASES ARE TO BE SPECIFIED?
2


 SPECIFY TEST CASE      1

 ENTER VALUES FOR
 N      ,
5

 ENTER      5 VALUES FOR ARRAY A
1 2 3 4 5


  TEST CASE NUMBER     1
 PARAMETERS ON INPUT
 N         =           5
 PARAMETERS ON OUTPUT
 A         (  1)=            1
 A         (  2)=            2
 A         (  3)=            3
 A         (  4)=            4
 A         (  5)=            5
   THE RAW PROGRAM TOOK     19 STEPS TO EXECUTE THIS TEST CASE
 HIT RETURN TO CONTINUE


 PLEASE VERIFY THAT DATA IS CORRECT
 TYPE A YES OR NO    ****
YES


 SPECIFY TEST CASE      2

 ENTER VALUES FOR
 N      ,
5
 ENTER      5 VALUES FOR ARRAY A
99 -99 -55 0 50


  TEST CASE NUMBER     2
 PARAMETERS ON INPUT
 N         =           5
```

```
 A         (  1)=          99
 A         (  2)=         -99
 A         (  3)=         -55
 A         (  4)=           0
 A         (  5)=          50
 PARAMETERS ON OUTPUT
 A         (  1)=         -99
 A         (  2)=         -55
 A         (  3)=           0
 A         (  4)=          50
 A         (  5)=          99
   THE RAW PROGRAM TOOK     51 STEPS TO EXECUTE THIS TEST CASE
 HIT RETURN TO CONTINUE

 PLEASE VERIFY THAT DATA IS CORRECT
 TYPE A YES OR NO    ****
YES



  WHAT NEW TYPES OF MUTANTS ARE TO BE CONSIDERED ?
ALL

  MUTATION PHASE

  POST RUN PHASE

 NUMBER OF TEST CASES =    2

 NUMBER OF LIVE MUTANTS =    31

 NUMBER OF MUTANTS =      240

 PERCENTAGE OF ELIMINATED MUTANTS =    87.08
```

```
 MUTANT TYPES AND LIVE MUTANTS PROFILES

 TYPE MUTANTS LIVE* TYPE MUTANTS LIVE* TYPE MUTANTS LIVE*
 ALD     1   0*   CRP    16   4*   SVR    42   3*   SFC    32   6*
 CFS    30   2*   CFA    12   0*   SFA    24   0*   AFC     8   2*
 AFS    12   0*   AOR    12   0*   ROR    15   5*   GLR     4   3*
 PAN    16   1*   CSD     1   0*   RSR    15   5*

 MUTANT ELIMINATION METHOD PROFILE

 METHOD     COUNT* METHOD     COUNT* METHOD     COUNT*
 TIMED-OUT     34* REF UNDVAR    47* SUBSCR RNG    38*
 ARTH FAULT     0* RDONLY VAR     0* TRAP STMT     15*
 EQUIV          0* ZERO DIV       0* WRONG ANS     75*


 POST RUN RESULTS
HELP
   COMMANDS CAN USUALLY BE ABBRIVIATED TO
         TWO LETTERS, COMMANDS ARE AS FOLLOWS :
     HELP       - DISPLAY THIS HELP PAGE (CANNOT ABBRIV.)
     KILL       - ABORT THE CURRENT RUN (CANNOT ABBRIV.)
     PROGRAM    - TYPE THE PROGRAM BEING MUTATED
     TESTCASE N- TYPE THE TESTCASE N
     MUTANTS    - TYPE ALL THE LIVE MUTANTS
     MUTANTS (KEYWORD) (KEYWORD) ... (KEYWORD)
                 - TYPE ONLY MUTANTS OF THE SPECIFIED TYPE
     MUTANTS SELECT - SELECT THE MUTANTS MENU STYLE
     MUTANTS KEYWORDS - SEE THE KEYWORDS FOR MUTANTS TYPES
     HEADER  - DISPLAY THE PIMS RUN HEADER
     CORRECTNESS - DISPLAY THE METHOD OF DETERMINING
          CORRECTNESS
     RESULTS - DISPLAY THE RESULTS FOR MUTANTS CREATED
          IN THIS RUN
     STATUS  - DISPLAY THE STATUS OF ALL MUTANTS (INCLUDING
          PREVIOUS RUNS)
     HALT       - STOP THE CURRENT PIMS RUN
     LOOP    -  ITERATE THE CURRENT RUN
     OUTPUT TESTCASES - JUST THAT,
     OUTPUT MUTANTS - OUTPUT ALL LIVE MUTANTS
     OUTPUT MUTANTS (KEYWORD) (KEYWORD) ... (KEYWORD)
          OUTPUT ONLY MUTANTS OF THE INDICATED TYPE
     OUTPUT MUTANTS RANDOM - OUTPUT ONE RANDOM MUTANT OF
          EACH TYPE
     OUTPUT MUTANTS RANDOM (KEYWORD) (KEYWORD) (KEYWORD)


 POST RUN RESULTS
HALT
```

BIBLIOGRAPHY

[1] C.V.Ramamoorthy, S.F.Ho, and W.T.Chen, "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering SE-2,4 (Dec 1976), pp 293-300.

[2] W.E.Howden, "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers C-24,5 (May 1975), pp 554-560.

[3] W.E.Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering SE-2,3 (Sept 1976), pp 208-214.

[4] J.B.Goodenough and S.L.Gerhart, "Towards a Theory of Test Data Selection", IEEE Transactions on Software Engineering SE-1,2 (June 1975), pp 156-173.

[5] J.C.Huang, "An Approach to Program Testing", Computing Surveys 7,3 (Sept 1975), pp 113-128.

[6] E.F.Miller and R.A.Melton, "Automated Generation of Testcase Datasets", in Procedings of the First International Conference on Reliable Software, SIGPLAN Notices 10,6 (June 1975), pp 51-58.

[7] L.Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering SE-2,3 (Sept 1976), pp 215-222.

[8] J.King, "Symbolic Execution and Program Testing", Communications of the ACM 19,7 (July 1976), pp 385-394.

[9] R.London, "The Current State of Proving Programs Correct", in Procedings of the ACM National Conference, 1972, ACM, New York, pp 39-46.

[10] S.Hantler and J.King, "An Introduction to Proving the Correctness of Programs", Computing Surveys 8,3 (Sept 1976), pp 331-353.

[11] E.A.Youngs, "Human Errors in Programming", International Journal of Man Machine Studies 6 (1974), pp 361-376.

[12] B.Boehm, "Software Design and Structuring", in Practical Strategies for Developing Large Software Systems, Horowitz (Editor), Addison-Wesley, 1975, pp 103-128.

[13] R.DeMillo, R.Lipton,and F.Sayward, "Hints on Test Data Selection", IEEE Computer, vol. 11, no. 4, April 1978, pp 34-41.

[14] Special Issue: Programming, ACM Computing Surveys 6,4 (Dec. 1974), pp 209-319

[15] R.DeMillo, R.Lipton, F.Sayward, "PROGRAM MUTATION: A Method of Determining Test Data Adequacy", State of the Art: Program Testing, SRA/INFOTECH, 1979.

[16] R. DeMillo, F. Sayward, "Program Mutation as a Tool for Managing Large Software Development," Trans. 36th Meeting for Quality Control.

[17] SOFTWARE TOOLS SUBSYSTEM USER'S GUIDE, (GIT-ICS-78/02), Georgia Institute of Technology, September, 1978.

[18] FORTRAN PROGRAMMER'S GUIDE, PDR3057, Prime Computer, Incorporated, Framingham, Massachusetts, November, 1977.

[19] THE NEW USER'S GUIDE TO EDITOR AND RUNOFF, PDR3104, Prime Computer, Incorporated, Framingham, Massachusetts, November, 1977.

[20] J.Burns, "Stability of Test Data From Program Mutation", DIGEST FOR THE WORKSHOP ON SOFTWARE TESTING AND TEST DOCUMENTATION, Ft. Lauderdale, Florida, 1978, pp 324-334.